Combining McCabe IQ with Fuzz Testing

Introduction

Fuzz testing, or fuzzing, is a black-box testing technique that has recently leapt to prominence as a quick and cost effective method for uncovering security bugs. Over 70% of security vulnerabilities Microsoft patched in 2006 were found by fuzzing[1]. This approach involves randomly mutating well-formed inputs and testing an application against the resulting data. Fuzzing is able to cover the most exposed and critical attack surfaces in a system and identify common errors and potential vulnerabilities quickly and cost-effectively. Although fuzz testing tools can be remarkably effective, their ability to discover bugs on low probability program paths is inherently limited. Many current code coverage tools are inadequate and inefficient for vulnerability analysis. A recent experiment in the Secure Systems Lab at the Technical University Vienna demonstrated that the system was exploring multiple paths in only a small fraction of malware samples in the evaluation set [2].This application note will detail how leveraging static and dynamic path analysis will improve fuzz testing and software security.

Is Complexity a Factor?

Being cognizant of control flow paths is nothing new to security experts. In an article published in *MSDN Magazine* entitled "CODE REVIEWS: Find and Fix Vulnerabilities Before Your Application Ships" [3], Michal Chmielewski, Neill Clift, Sergiusz Fonrobert and Tomasz Ostwald.[4] detail how to perform source code security reviews and identify high risk code paths for review. For each vulnerability candidate, a reviewer follows up all code paths in order to determine whether the coding error actually represents a vulnerability or processing data that can be controlled by an attacker over a security boundary.

In his book "Security Metrics: Replacing Fear Uncertainty and Doubt",: Andrew Jaquith wrote the following: "Beyond the direct security issues that scanning products can find and enumerate in code modules, organizations should also consider the broader issue of code complexity. Both Bruce Schneier and Dan Geer are fond of pointing out that 'complex systems fail complexly'. Modern applications are typically complex code edifices constructed with care, built for extensibility, and possessed of more layers than a Herman Melville novel. This is not necessarily a bad thing, but it makes it harder to find and eliminate the root causes of security problems. Thus, if complexity contributes to insecurity, we ought to devise methods for measuring code complexity as a leading indicator of future security problems. Cyclomatic complexity is the right metric for measuring control flow density on a per-method and per-entity (or per-class) basis. Because security flaws are, at least some of the time, implementation-related, cyclomatic complexity metrics can help predict which classes/methods in an application might experience flaws. Code metrics such as vulnerability density and cyclomatic complexity provide raw measures of how secure and reliable code modules are likely to be."[5]

Cyclomatic complexity has also been mentioned as a possible detection method for particularly nasty bugs. In the paper " The Little Hybrid Web Worm that Could," Billy Hoffman, Lead Researcher at SPI Dynamics, and John Terrill, Co-founder of Enterprise Technology, had this to say: "One possible detection method is to examine the Cyclomatic Complexity or McCabe Complexity of a piece of arbitrary JavaScript code. The overall complexity diagram and number of closed loops should remain almost identical regardless of the number of mutations performed on the code. This follows since our mutations change the syntax of the code but not the underlying functionality then the complexity of that functionality should remain the same."[6] The authors are investigating whether a complexity diagram alone is capable of uniquely identifying web malware."[7]

Did the Fuzz Testing Catch Everything?

Fuzz testing can be easily automated and conducted on a continuous basis, but it operates in at least a partially random manner and may have problems with reaching deeper parts of the code. In most cases it is relatively easy to conduct basic fuzzing, yet it is much more difficult to achieve complete coverage of the associated critical code paths.

The main problem with fuzzing to find program faults is that it generally only finds very simple faults. The problem itself is exponential and every fuzzer takes shortcuts to find something interesting in a timeframe that a human cares about. Most primitive black box fuzzers get poor code coverage; for example, if the input includes a checksum which is not properly updated to match other random changes, only the checksum validation code will be verified. Code coverage tools are often used to estimate how "well" a fuzzer works or how much of the critical logic has been verified. It is important to use an intelligent black box fuzz testing methodology capable of crafting inputs which force an application to execute specific dependent portions of its control flow graph.

Our automated fuzz test process uses both static and dynamic source code analysis in conjunction with fuzz tests to increase the effectiveness of their vulnerability detection program. This is done by using the static control flow analysis as an attack map and by running all fuzz tests on an instrumented version of source code to track the code coverage of the fuzz tests. After the fuzz test code coverage is reviewed, additional test path information is used to tweak the fuzz tests to execute additional portions of critical logic paths. The McCabe IQ slice functionality can be used to evaluate tainted and untainted data execution traces through the source code subtree invocations and cyclomatic control flow graphs. This is similar to what is done internally in Dynamic Data Flow Analysis (DDFA) using the Broadway static data flow analysis and error checking system, developed by UT Austin and Southwest Research Institute[8].

Microsoft is using their FuzzGuru framework, as an approach to fuzzing and has a tight integration with code coverage tools. A recently published paper by Microsoft Research, "Automated Whitebox Fuzz Testing"[9], details Microsoft's method of fuzzing. Their approach seems to provide good results and is very path-sensitive. Using path-sensitive code coverage and static analysis to monitor, plan and augment fuzz testing is detailed at Microsoft Research. It is called "white box fuzz testing" and, although the Microsoft tools are using binaries or byte code for their analysis, the same systematic method can, and should, be used on source code. The idea is to use control flow graphs and subtree invocation diagrams to understand the internals of the code and to think in terms of "producing new inputs, which cause the program to follow different control paths. This process is repeated with help of a code coverage monitoring tool to find defects as fast as possible."

Another interesting approach to fuzzing that is cyclomatic path-sensitive has been outlined by Mu Dynamics Research Labs at the CanSecWest 2008 Conference. It is called Fieldomatic Complexity[10]. Field's are the fundamental units of protocols (network or file formats). The linkage information between the Field's and across messages is a powerful way to infer the cyclomatic complexity of the code that parses these messages. When generating test cases (fuzzing being one type), we can leverage these structural and semantic linkages to generate systematic constraint violations that ultimately exercise the various branches taken in the parser.
Often fuzzing bugs are in "clean-up" code paths: leaks, synchronization and timing issues. McCabe IQ can be used in conjunction with fuzz test tools to create a total testing solution. To complete the test cycle, the testing suites created by the fuzz test tools need to be analyzed to determine if they are complete and if any necessary modifications need to be made to assure full testing of the software. Integration of McCabe IQ with fuzz test tools will achieve a greater confidence in the software testing process.

McCabe IQ

McCabe IQ enables software engineers and managers to save time and reduce costs through efficient and effective allocation of testing resources. McCabe IQ will augment the fuzz testing process by locating the error prone modules and highlighting the untested paths in the code. McCabe IQ provides these capabilities in the following manner:

Basis Path Testing Methodology

McCabe IQ determines test paths that need to be run to fully test the software. McCabe IQ identifies test paths based on the McCabe Basis Path Testing Methodology. Basis Path testing demands that every outcome of a decision be tested independently. The number of tests needed to fully test a module is equal to the McCabe Cyclomatic Complexity. Therefore, by keeping code structured and reducing complexity, a developer can cut down the amount of time needed to test the code. McCabe IQ calculates and displays the complexity of code as well as the test paths associated with that complexity. Both static and dynamic path information is reported from McCabe IQ.

Coverage Analysis

McCabe IQ also determines the path coverage of a testing effort, indicating graphically and textually what code has been executed and what tests remain to be run to complete basis path testing. McCabe IQ provides figures for basis path coverage, branch coverage, and line coverage.

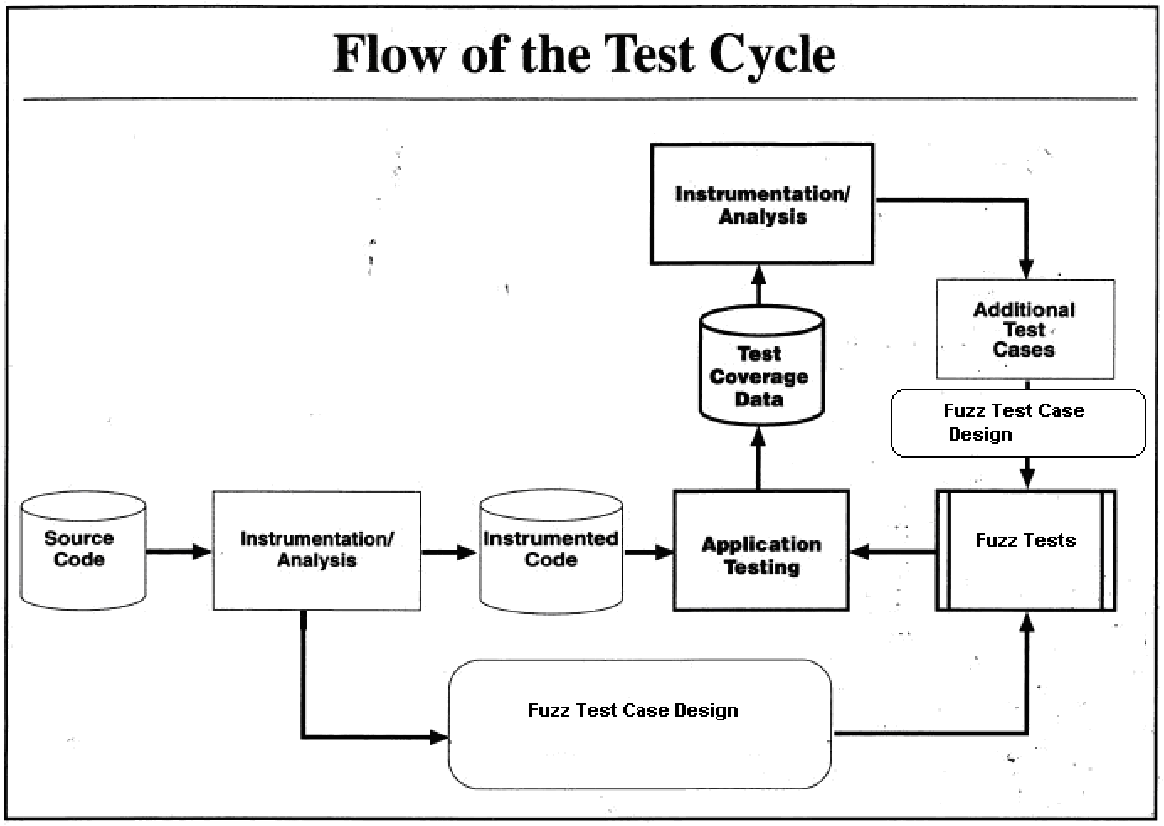

Figure 1. An example of the test flow cycle, including both fuzz test tools and McCabe IQ

The Testing Process

The software testing cycle is a seven-step process that includes the McCabe IQ and fuzz test tools *(see Figure 1).*

1. **All code in question should be analyzed to determine where the focus of the testing efforts should be using control flow graphs and Subtree Call Graphs.**

2. **Initial fuzz tests need to be developed.** Fuzz tests can be created based on the functionality of the software or can be requirements driven. The primary focus of these tests should be geared towards the more complex code in order to weed out potential bugs in the unstructured portions.

3. **The Code is then analyzed by McCabe IQ and an instrumented version of the source code is produced.** The instrumentation consists of flags and hooks inserted into the code allowing the coverage information to be mapped back into the McCabe IQ static control flow diagrams. The instrumented code is then compiled on the target system. The analyst can selectively instrument portions of the code after insights are gained from the static control flow analysis. The analyst may wish to focus fuzz testing resources and code coverage monitoring on specific subsets of the source code such as: of the state space most likely to be targeted by malicious users (e.g. a security analyst should examine parsing of logic for a packet received off of an open port more closely than GUI code accepting mouse input).

4. **The fuzz tests are then executed on the instrumented version of the code.**
   The instrumented code creates a trace file that contains the coverage data for the test suite. The trace file with test coverage data is imported into McCabe IQ. When the file is imported back into McCabe IQ, the information will be mapped onto the static analysis structure charts and unit level control flow diagrams. The tool will then calculate what tests remain to complete the desired coverage. Black box fuzzers often have difficulty achieving good code coverage and penetration depth into a program's control flow logic.

5. **Analysis is performed to determine which sections of code were tested by the test suite and where further testing is needed.** Additions can be made to the suites and the code should be tested. After running a fuzzing test for a first round it is important to inspect whether all code paths were covered. Fuzzing should traverse all basis paths that depend on untrusted data. All parsing code should be covered. All error handling and cleanup code should be covered. If there is no code to handle malformed data existing, code coverage will not help you.

   To add coverage, the analyst may need to: Extend test matrix and fuzzing tests for additional configurations of the feature, add additional fuzzing templates and/or extend the fuzz testing tool with custom fuzzers or elements.

   Input selection can then be based upon guided feedback concerning progress within the program logic being tested. In order to determine if the vulnerability is an exploitable threat, one must prove that it is reachable on the execution path given some user supplied input. The exact format of this input is dependent upon the control flow logic on the path between the packet acceptance and the basic block where the vulnerable API function is used.[11] Figure 1 provides a graphical illustration of this idea.

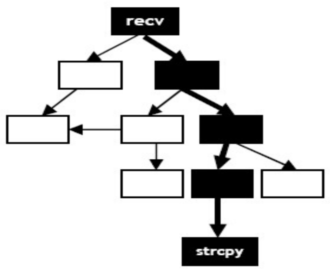

Figure 1. An idealized diagram of the input crafting problem (i.e. what input will cause the program to exercise the control flow logic on the path from the recv function to a potentially vulnerable strcpy()?)

6. **Metrics**. Basis Path, Branch, Boolean, Line and Basis Subtree test coverage metrics are produced after dynamic analysis of the source code. Every path, branch, line of code and integration subtree that was executed during the fuzz test will be evaluated. Static Analysis is available prior to running fuzz tests, and produces code comprehension metrics such as cyclomatic complexity, essential complexity, module design complexity and integration complexity. These static path metrics provides information on various characteristics of the code under test and can help fuzz testers evaluate the source code logic or algorithms contained within the codebase.

Benefits and Summary

By combining McCabe IQ with fuzz test tools in the above fashion, testers can gain confidence in their test and in their software. McCabe Software expects these processes to produce:

- Increased code coverage and penetration depth into a program's control flow logic from fuzzing
- Better predictions of fuzz testing validation effort
- Increased effectiveness of fuzz testing
- Extension of the reach of black box fuzz testing

This process gives assurance that the fuzz testing is thorough, makes complete coverage repeatable, and increases the reach and effectiveness of traditional black box fuzz testing.

References

[1] John Neystadt, Microsoft "Fuzzing in Microsoft and FuzzGuru Framework" OWASP May 2007
https://www.owasp.org/images/5/5b/OWASP_IL_7_FuzzGuru.pdf

[2] Andreas Moser, Christopher Kruegel, and Engin Kirda, Secure Systems Lab, Technical University Vienna, "Exploring Multiple Execution Paths for Malware Analysis"
http://www.cs.ucsb.edu/~chris/doc/oakland07_explore.pdf

[3] Michal Chmielewski, Neill Clift, Sergiusz Fonrobert and Tomasz Ostwald, "CODE REVIEWS: Find and Fix Vulnerabilities Before Your Application Ships" MSDN Magazine
http://msdn.microsoft.com/en-us/magazine/cc163312.aspx

[4] Michal Chmielewski, Neill Clift, Sergiusz Fonrobert and Tomasz Ostwald, "CODE REVIEWS: Find and Fix Vulnerabilities Before Your Application Ships" MSDN Magazine
http://msdn.microsoft.com/en-us/magazine/cc163312.aspx

[5] Andrew Jacquith, "Security Metrics: Replacing Fear, Uncertainty and Doubt" Addison-Wesley Professional March 26, 2007
http://safari.awprofessional.com/9780321349989

[6] Billy Hoffman, SPI Dynamics, John Terrill Co-founder Enterprise Management Technology " The Little Hybrid Web Worm that Could"
https://www.blackhat.com/presentations/bh-usa-07/Hoffman_and_Terrill/Whitepaper/bh-usa-07-hoffman_and_terrill-WP.pdf

[7] Billy Hoffman, SPI Dynamics, John Terrill Co-founder Enterprise Management Technology " The Little Hybrid Web Worm that Could"
https://www.blackhat.com/presentations/bh-usa-07/Hoffman_and_Terrill/Whitepaper/bh-usa-07-hoffman_and_terrill-WP.pdf

[8] Steve Cook, Southwest Research Institute
Calvin Lin, Ph.D., University of Texas at Austin
Walter Chang, University of Texas at Austin
"Securing Legacy C Applications Using Dynamic Data Flow Analysis"
http://www.stsc.hill.af.mil/crosstalk/2008/09/0809CookLinChang.html

[9] Patrice Godefroid; Michael Y. Levin; David Molnar, "Automated Whitebox Fuzz Testing"
MS Research
ftp://ftp.research.microsoft.com/pub/tr/TR-2007-58.pdf

[10] Mu Dynamics Research Labs "Fieldomatic Complexity"--CanSecWest 2008 Conference
http://labs.mudynamics.com/2008/05/23/fieldomatic-complexity/#more-103

Patrice Godefroid, Peli de Halleux, Aditya V. Nori, Sriram K. Rajamani,
Wolfram Schulte, and Nikolai Tillmann "Automating Software Testing Using Program Analysis""
Microsoft Center for Software Excellence
http://research.microsoft.com/users/pg/public_psfiles/ieeesw2008.pdf

Information Assurance Technology Analysis Center (IATAC)
Data and Analysis Center for Software (DACS)
Joint endeavor by IATAC with DACS
"Software Security Assurance State-of-the-Art Report (SOAR)"
July 31, 2007
https://www.thedacs.com/techs/abstracts/abstract.php?dan=347617

[11] Sherri Sparks, Shawn Embleton, Ryan Cunningham, Cliff Zou, University of Central Florida
"Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting"
http://www.cs.ucf.edu/~czou/research/EvolutionaryInputCrafting-ACSAC07.pdf